A Framework For Simple Sorting Algorithms On Parallel Disk Systems

Sanguthevar Rajasekaran
Dept. of CISE, University of Florida
Gainesville FL 32611

Abstract

In this paper we present a simple parallel sorting algorithm and illustrate its application in general sorting, disk sorting, and hypercube sorting. The algorithm (called the (l, m)-mergesort (LMM)) is an extension of the bitonic and odd-even mergesorts.

Literature on parallel sorting is abundant. Many of the algorithms proposed, though being theoretically important, may not perform satisfactorily in practice owing to large constants in their time bounds. The algorithm to be presented in this paper has the potential of being practical.

We present an application for the parallel disk sorting problem. The algorithm is asymptotically optimal (assuming that N is a polynomial in M, where N is the number of records to be sorted and M is the internal memory size). The underlying constant is very small. This algorithm performs better than the disk-striped mergesort (DSM) algorithm when the number of disks is large. Our implementation is as simple as that of DSM (requiring no fancy data structures or prefetch techniques.)

As a second application, we prove that we can get a sparse enumeration sort on the hypercube that is simpler than that of the classical algorithm of Nassimi and Sahni [16]. We also show that Leighton's columnsort algorithm is a special case of LMM.

1 Introduction

Sorting is one of the most widely studied problems in computing. Numerous asymptotically optimal sequential algorithms have been discovered. Asymptotically optimal algorithms have been presented for various parallel models as well. The classic algorithm of Batcher [5] was nearly optimal with processor and time bounds of n and $O(\log^2 n)$, respectively, to sort n numbers. The paper of Ajtai, Komlós and Szemerédi [3] gave the first asymptotically optimal logarithmic time deterministic parallel algorithm for sorting. Reischuk's algorithm for the PRAM [23] and the Flashsort of Reif and Valiant [22] were asymptotically optimal randomized algorithms. Some of the follow-up algorithms include Leighton's column sort [15] and Cole's optimal deterministic algorithm for the PRAM [7]. These sorting results have been employed in the design of numerous other parallel algorithms also.

Since sorting is a fundamental problem, it is imperative to have efficient algorithms to solve it. Though the literature on sorting is vast, many of these algorithms have huge constants in their run times, making them inferior in practice to asymptotically inferior algorithms. For a survey of parallel sorting algorithms the reader is referred to [21].

This paper is motivated by a desire to seek practical algorithms. In particular, we are interested in the development of sorting algorithms that will have small underlying constants. We introduce a variant of the bitonic and odd-even mergesort algorithms called the (l, m)-mergesort (LMM). To demonstrate its usability, we present two illustrative applications.

The first application is for the parallel disk sorting problem. This problem also has been extensively studied on several related models. The model we use is the one suggested by Vitter and Shriver in their pioneering paper [25]. A known lower bound for the number of I/O read steps for parallel disk sorting is[1] $\Omega\left(\frac{N}{DB}\left\lceil\frac{\log(N/B)}{\log(M/B)}\right\rceil\right)$. Here N is the number of records to be sorted and M is the internal memory size. Also, B is the

[1]Throughout this paper we use log to denote logarithms to the base 2 and ln to denote natural logarithms.

block size and D is the number of parallel disks used. There exist several asymptotically optimal algorithms that make $O\left(\frac{N}{DB}\left[\frac{\log(N/B)}{\log(M/B)}\right]\right)$ I/O read steps (see e.g., [17, 1, 4]).

Our implementation results in an asymptotically optimal algorithm under the assumption that N is a polynomial in M. This assumption is easily met in practice. For instance in today's SMP market, M is typically of the order of megabytes for individual processors. Disk sizes are of the order of gigabytes. So, it is perhaps safe to assume that $N \leq M^3$. In particular, the number of I/O read steps needed in our algorithm is no more than $\frac{N}{DB}\left[\frac{\log(N/M)}{\log\left(\min\{\sqrt{M}, M/B\}\right)}+1\right]^2$. This complexity bound is not dependent on the above assumptions. If $N = M^c$, for some constant c, and B is small (e.g., M is a polynomial in B) then this bound is $\Theta\left(\frac{N}{DB}\left[\frac{\log(N/B)}{\log(M/B)}\right]\right)$.

Our implementation is very simple and requires no fancy data structures. The internal memory requirement is only $3DB$. We illustrate with examples that when D is large, LMM performs better than DSM. We also believe that when D is large LMM has the potential of comparing favorably to the simple randomized algorithm (SRM) proposed by Barve, Grove, and Vitter [6].

In addition, we prove that the LMM algorithm can be used to solve the sparse enumeration sort on the hypercube. Such an implementation is conceptually simpler than Nassimi and Sahni's algorithm [16].

In Section 2 we give a description of the (l, m)-mergesort and prove its correctness. In Section 3 we present details of our parallel disk sorting application. Section 4 compares the three algorithms DSM, SRM, and LMM. Section 5 is devoted to the application of LMM to sparse enumeration sort. In section 6 we relate LMM with the column sort algorithm. Section 7 concludes the paper.

2 The (l, m)-merge Sort (LMM)

The *odd-even mergesort* [14, 12] algorithm employs the odd-even merge algorithm repeatedly to merge two sequences at a time. The *odd-even mergesort* [14, 12], the *bitonic sort* [5], and the *periodic balanced mergesort* [10] are all very similar. We use the term *odd-even mergesort* to refer to these algorithms. These algorithms have a common theme (up to some slight variations).

Let $k_1, k_2, \ldots, k_n$ be a given sequence of n keys. Assume that $n = 2^h$ for some integer h. The odd-even mergesort begins by forming $\frac{n}{2}$ sorted sequences of length two each.

Next, it merges two sequences at a time so that at the end $\frac{n}{4}$ sorted sequences of length 4 each will remain. This process of merging is continued until only two sequences of length $\frac{n}{2}$ each are left. Finally these two sequences are merged.

Algorithm Odd-Even Merge

Step 1. Let $U = u_1, u_2, \ldots, u_q$ and $V = v_1, v_2, \ldots, v_q$ be the two sorted sequences to be merged. *Unshuffle* U into two, i.e., partition U into two: $U_{odd} = u_1, u_3, \ldots, u_{q-1}$ and $U_{even} = u_2, u_4, \ldots, u_q$. Similarly partition V into V_{odd} and V_{even}.

Step 2. Now recursively merge U_{odd} with V_{odd}. Let $X = x_1, x_2, \ldots, x_q$ be the result. Also merge U_{even} with V_{even}. Let $Y = y_1, y_2, \ldots, y_q$ be the result.

Step 3. *Shuffle* X and Y, i.e., form the sequence: $Z = x_1, y_1, x_2, y_2, \ldots, x_q, y_q$.

Step 4. Perform one step of *compare-exchange operation*, i.e., sort successive subsequences of length two in Z. In other words, sort y_1, x_2; sort y_2, x_3; and so on. The resultant sequence is the merge of U and V.

One can use the zero-one principle to prove the correctness of the above merge algorithm (see e.g., [11, 14]). An extension of this idea has been employed by Thompson and Kung [24] to design an asymptotically optimal algorithm for sorting on the mesh model of parallel computing. Their algorithm, called the s^2-way merge, partitions the given n-element sequence to be sorted into s^2 evenly sized parts (for some appropriate function s of n), recursively sorts each part, and merges the s^2 sorted parts. In order to merge s^2 sorted sequences, the sequences are unshuffled into two components, namely the odd and even components. Each component is merged recursively, the results are shuffled, and some local sorting is done. Effectively, the problem of merging s^2 sequences is reduced to two subproblems, where each subproblem is that of merging s^2 subsequences. The subsequences now will be of length one-half of the length of the original sequences. The base case is that of merging s^2 sequences of length one each. This case is handled by a different algorithm.

LMM is a generalization of the odd-even mergesort and s^2-way mergesort. Here also the sequence to be sorted is partitioned into l parts (for some appropriate l). Each part is recursively sorted. To merge these l sequences, the sequences are unshuffled into m components (instead of two). More details follow.

Algorithm LMM

Step 1. Let $K = k_1, k_2, \ldots, k_n$ be the sequence to be sorted. Partition K into l evenly sized parts. Let these parts be $K_i = k_{(i-1)n/l+1}, k_{(i-1)n/l+2}, \ldots, k_{in/l}$, for $i = 1, 2, \ldots, l$[2]. Sort each part recursively. Let the sorted sequences be $U_1, U_2, \ldots, U_l$.

Step 2. Merge $U_1, U_2, \ldots, U_l$ using **Algorithm** (l, m)-merge.

Now we describe the underlying merge algorithm. Figure 1 illustrates the steps involved in this algorithm.

Algorithm (l, m)-merge

Step 1. Let the sequences to be merged be $U_i = u_i^1, u_i^2, \ldots, u_i^r$, for $1 \leq i \leq l$. If r is *small* use a base case algorithm (e.g., any sorting algorithm). Otherwise, unshuffle each U_i into m parts. In particular, partition U_i into $U_i^1, U_i^2, \ldots, U_i^m$, where $U_i^1 = u_i^1, u_i^{1+m}, \ldots$; $U_i^2 = u_i^2, u_i^{2+m}, \ldots$; and so on.

Step 2.
Recursively merge $U_1^j, U_2^j, \ldots, U_l^j$, for $1 \leq j \leq m$. Let the merged sequences be $X_j = x_j^1, x_j^2, \ldots, x_j^{lr/m}$, for $1 \leq j \leq m$.

Step 3. Shuffle $X_1, X_2, \ldots, X_m$, i.e., form the sequence $Z = x_1^1, x_2^1, \ldots, x_m^1, x_1^2, x_2^2, \ldots, x_m^2, \ldots, x_1^{lr/m}, x_2^{lr/m}, \ldots, x_m^{lr/m}$.

Step 4. It can be shown that at this point the length of the 'dirty sequence' (i.e., unsorted portion) is no more than lm. But we don't know where the dirty sequence is located. We can cleanup the dirty sequence in many different ways. One way is described below.

Call the sequence of the first lm elements of Z as Z_1; the next lm elements as Z_2; and so on. In other words, Z is partitioned into $Z_1, Z_2, \ldots, Z_{r/m}$. Sort each one of the Z_i's. Followed by this merge Z_1 and Z_2; merge Z_3 and Z_4; etc. Finally merge Z_2 and Z_3; merge Z_4 and Z_5; and so on.

[2] Assume without loss of generality that n/l is an integer

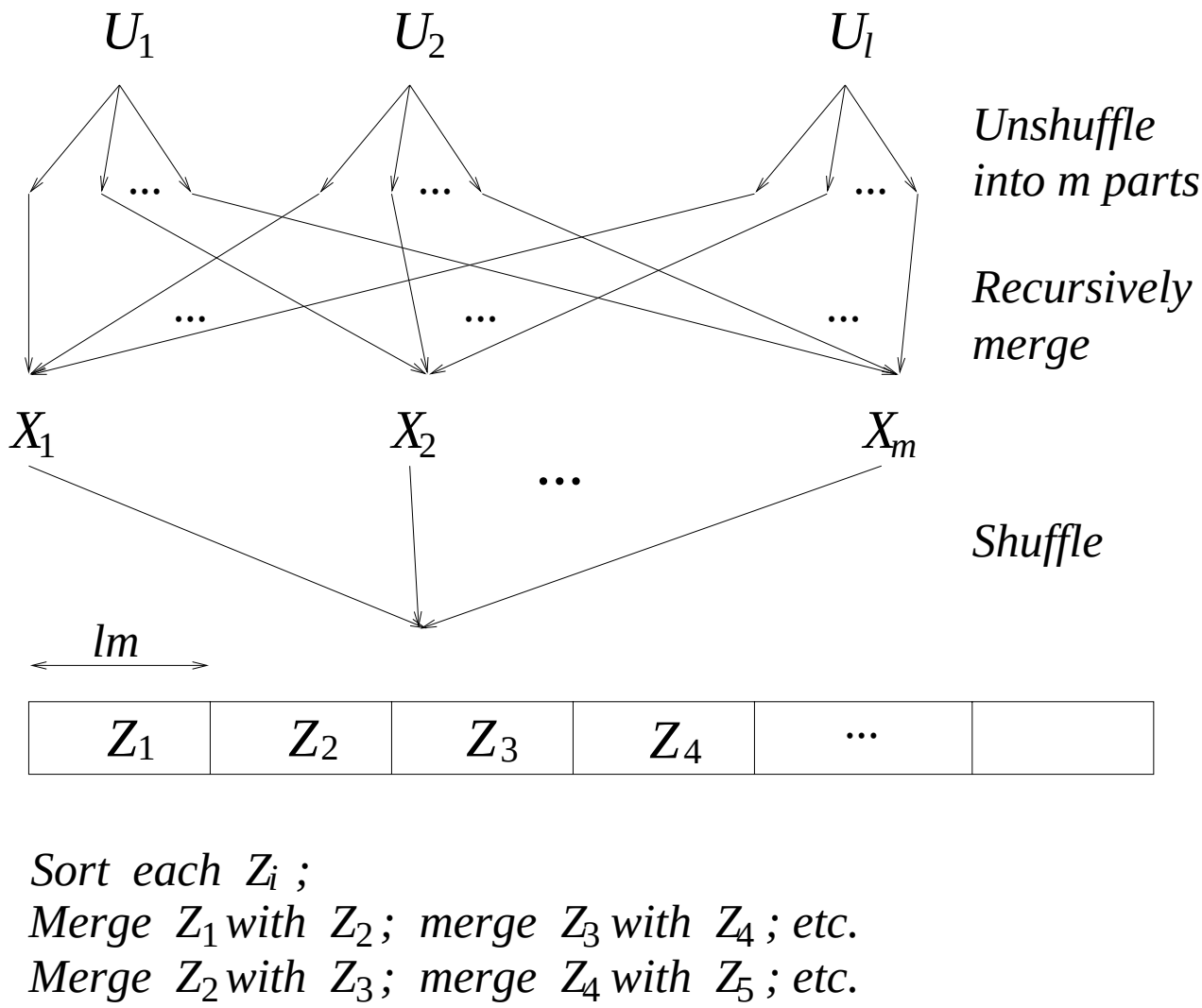

Figure 1: Algorithm (l, m)-merge

Proof of correctness. Note that it suffices to prove the correctness of the merge algorithm since the sorting algorithm functions by repeatedly invoking the merge algorithm. We prove the correctness of **Algorithm** (l, m)-merge using the zero-one principle. Since the algorithm is oblivious, the zero-one principle holds. Assume that the sequence to be sorted consists of only zeros and ones.

Let the number of zeros in U_i be z_i, for $1 \leq i \leq l$. The minimum number of zeros contributed by any U_i to any X_j $(1 \leq i \leq l;\ 1 \leq j \leq m)$ is $\lfloor \frac{z_i}{m} \rfloor$. The maximum number of zeros contributed by any U_i to any X_j is $\lceil \frac{z_i}{m} \rceil$. Thus the minimum number of zeros in any X_j is $z_{min} = \sum_{i=1}^{l} \lfloor \frac{z_i}{m} \rfloor$. The maximum number of zeros in any X_j is $z_{max} = \sum_{i=1}^{l} \lceil \frac{z_i}{m} \rceil$.

The difference between z_{max} and z_{min} can be at most l. This in turn means that when the X_j's are shuffled, the length of the dirty sequence (i.e., the unsorted portion) can be at most lm. The fact that Step 4 cleans up the dirty sequence is also easy to see. Intuitively, at the end of Step 3, the sequence is almost in sorted order. In particular, every element is at a distance of at most lm from its final position in sorted order. That's why the local sorting done in Step 4 helps in obtaining sorted order. This completes the proof of correctness. □

Observation. The odd-even mergesort is nothing but LMM with $l = m = 2$. Thompson and Kung's s^2-way merge sort is a special case of LMM with $l = s^2$ and $m = 2$ [24].

3 Parallel Disk Sorting

The problem of external sorting has been widely explored owing to its paramount importance. Memory is expensive and in some cases limited due to architectural considerations. With the widening gap between processor speeds and disk access speeds, the I/O bottleneck has become critical. Parallel disk systems have been introduced to alleviate this bottleneck.

Several models for parallel disks have been investigated. The model employed in this paper is the one introduced by Vitter and Shriver [25]. In this model there are D distinct and independent disk drives. In one parallel I/O operation, each disk transmits one block of data. A block consists of B records. If M is the internal memory size, then one usually requires that $M \geq 2DB$ to overlap I/O with local computations. For the algorithms presented in this paper, a choice of $M = 3DB$ suffices. Of this, only

$2DB$ amount of memory is used to store data to be currently operated on. In the other portion, we store prefetched data in order to overlap computation and data access. From hereon, M is used to refer to only DB.

The problem of disk sorting was first studied by Aggarwal and Vitter in [2]. In the model they considered, each I/O operation results in the transfer of D blocks each block having B records. A more realistic model was envisioned in [25]. Several asymptotically optimal algorithms have been given for sorting on this model. Nodine and Vitter's optimal deterministic sorting algorithm [18] involves solving certain matching problems. Aggarwal and Plaxton's optimal algorithm [1] is based on the Sharesort algorithm of Cypher and Plaxton [9]. Vitter and Shriver gave an optimal randomized algorithm for disk sorting [25]. All these results are highly nontrivial and theoretically interesting. However, the underlying constants in their time bounds are high as can be seen from an analysis of their algorithms. An implementation of the radix sort on parallel disks has been given in [8].

The disk-striped mergesort (DSM) [6] has the advantages of simplicity and a small constant. Data accesses made by DSM is such that at any I/O operation, the same portions of the D disks are accessed. This has the effect of having a single disk which can transfer DB records in a single I/O operation. An $\frac{M}{DB}$-way mergesort is employed by this algorithm. To start with, initial runs are formed in one pass through the data. At the end the disk has N/M runs each of length M. Next, $\frac{M}{DB}$ runs are merged at a time. Blocks of any run are uniformly striped across the disks so that in future they can be accessed in parallel utilizing the full bandwidth. Each phase of merging involves one pass through the data. There are $\frac{\log(N/M)}{\log(M/DB)}$ phases and hence the total number of passes made by DSM is $\frac{\log(N/M)}{\log(M/DB)}$. In other words, the total number of I/O read operations performed by the algorithm is $\frac{N}{DB}\left(1+\frac{\log(N/M)}{\log(M/DB)}\right)$. The constant here is just 1.

The known lower bound on the number of passes for parallel disk sorting is $\Omega\left(\frac{\log(N/B)}{\log(M/B)}\right)$. If one assumes that N is a polynomial in M and that B is small (which are readily satisfied in practice), the lower bound simply yields $\Omega(1)$ passes. All the abovementioned optimal algorithms make only $O(1)$ passes. So, the challenge in the design of parallel disk sorting algorithms is in reducing this constant. If $M = 2DB$, the number of passes made by DSM is $1 + \log(N/M)$, which indeed can be very high.

Recently, several works have been done that deal with the practical aspects. Pai, Schaffer, and Varman [19] analyzed the average case performance of a simple merging algorithm, employing an approximate model of average case inputs. Barve, Grove,

and Vitter [6] have presented a simple randomized algorithm (SRM) and analyzed its performance. The analysis involves the solution of certain occupancy problems. The expected number R_{SRM} of I/O read operations made by their algorithm is such that

$$R_{SRM} \leq \frac{N}{DB}\left[1 + \frac{\ln(N/M)}{\ln kD}\frac{\ln D}{k \ln\ln D}\left(1 + \frac{\ln\ln\ln D}{\ln\ln D} + \frac{1+\ln k}{\ln\ln D} + O(1)\right)\right] \quad (1)$$

The SRM algorithm merges $R = kD$ runs at a time, for some integer k. When $R = \Omega(D \log D)$, the expected performance of their algorithm is optimal. However, in this case, the internal memory needed is $\Omega(BD \log D)$. They have also compared SRM with DSM through simulations and shown that SRM performs better than DSM.

The algorithm presented in this paper is asymptotically optimal under the assumptions that N is a polynomial in M and B is small. The algorithm is an application of the (l, m)-mergesort. The algorithm is as simple as DSM. We do not need any fancy data structures or prefetching techniques. The standard overlapping of computations and I/O operations can be done. The internal memory requirement is only $3DB$. We demonstrate with examples that our algorithm makes fewer passes than DSM when D is large.

Our algorithm merges R runs at a time, for some appropriate R. Since our algorithm is also based on merging in phases, we have to specify how the runs in a phase are stored across the D disks. Let the disks as well as the runs be numbered from zero. Each run will be striped across the disks. If $R \geq D$, the starting disk for the ith run is $i \bmod D$, i.e., the zeroth block of the ith run will be in disk $i \bmod D$; its first block will be in disk $(i+1) \bmod D$; and so on. This will enable us to access, in one I/O read operation, one block each from D distinct runs and hence obtain perfect disk parallelism. See Figure 2. If $R < D$, the starting disk for the ith run is $i\frac{D}{R}$. (Assume without loss of generality that D divides R.) Even now, we can obtain $\frac{D}{R}$ blocks from each of the runs in one I/O operation and hence achieve perfect disk parallelism.

In practice the value of B will be much less than M. For example, if $\frac{M}{B} > \sqrt{M}$, then the number of read passes made by our algorithm is no more than $\left(2\frac{\log(N/M)}{\log M} + 1\right)^2$. But for the sake of completeness, we also consider the case $\frac{M}{B} \leq \sqrt{M}$. In either case, we show that the number of read passes made by our algorithm is upper bounded by $\left[\frac{\log(N/M)}{\log\left(\min\{\sqrt{M}, M/B\}\right)} + 1\right]^2$. Like all the algorithms in the literature, our algorithm also forms initial runs of length M each in one read pass through the data. After this, the runs will be merged R at a time. Throughout, we use $T(u, v)$ to denote the number of

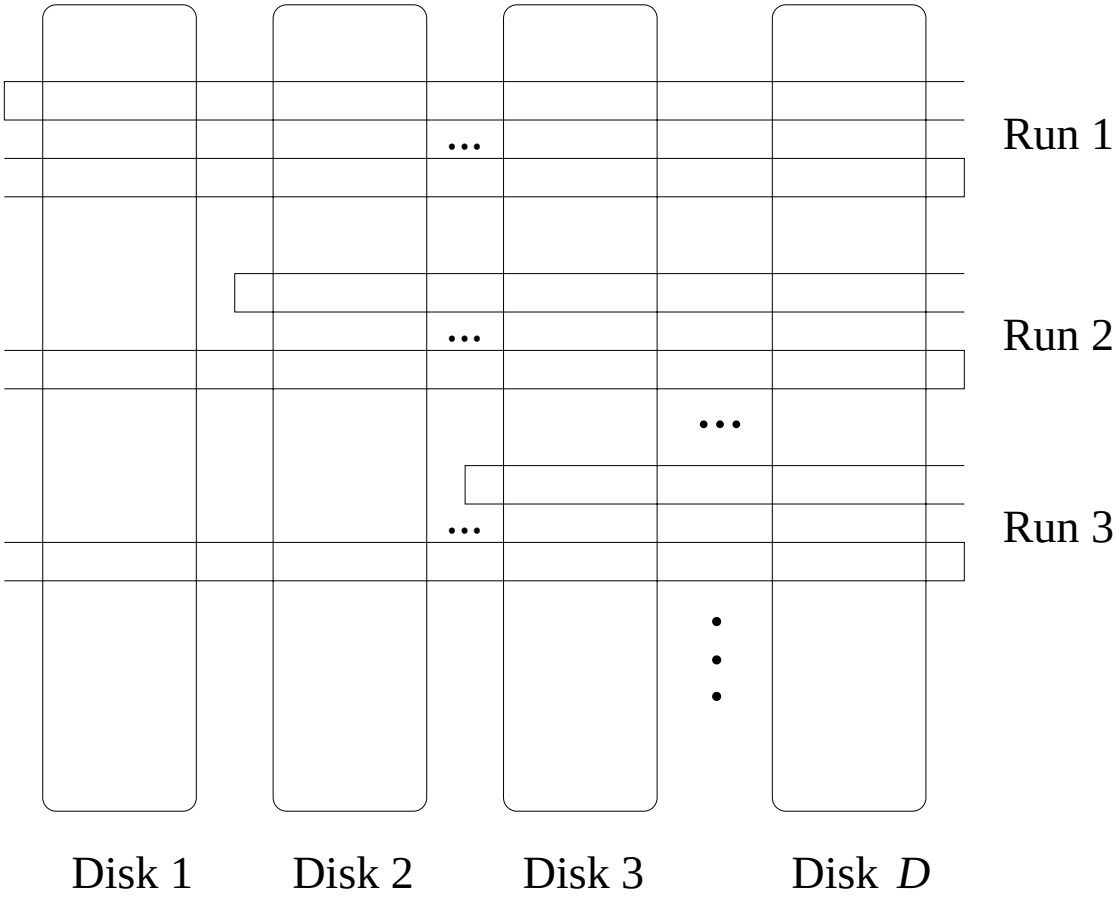

Figure 2: Striping of runs

read passes needed to merge u sequences of length v each.

3.1 Some Special Cases

We begin by looking at some special cases. Consider the problem of merging $\sqrt{M}$ runs each of length M, when $\frac{M}{B} \geq \sqrt{M}$. Here $R = \sqrt{M}$. This merging can be done using **Algorithm** (l, m)-merge with $l = m = \sqrt{M}$.

Let $U_1, U_2, \ldots, U_{\sqrt{M}}$ be the sequences to be merged. In Step 1, each U_i gets unshuffled into $\sqrt{M}$ parts so that each part is of length $\sqrt{M}$. This unshuffling can be done in one pass. In Step 2, we have $\sqrt{M}$ merges to do, where each merge involves $\sqrt{M}$ sequences of length $\sqrt{M}$ each. Observe that there are only M records in each merge and hence all the mergings can be done in one pass through the data. Step 3 involves shuffling and Step 4 involves cleaning up. The length of the dirty sequence is $(\sqrt{M})^2 = M$. These two steps can be combined and finished in one pass through the data. The idea is to have two successive Z_i's (c.f. **Algorithm** (l, m)-merge) (call these Z_i and Z_{i+1}) at any time in the main memory. We can sort Z_i and Z_{i+1} and merge them. After this Z_i is ready to be shipped to the disks. Z_{i+2} will then be brought in, sorted, and merged with Z_{i+1}. At this point Z_{i+1} will be shipped out; and so on.

Note that throughout we can maintain perfect disk parallelism. Thus we get:

Lemma 3.1 *$T(\sqrt{M}, M) = 3$, if $\frac{M}{B} \geq \sqrt{M}$.*

Now consider the case of merging $\frac{M}{B}$ runs each of length M, when $\frac{M}{B} < \sqrt{M}$. To solve this problem, employ **Algorithm** (l, m)-merge with $l = m = \frac{M}{B}$. Note that we have assumed $M = DB$.

Let the sequences to be merged be $U_1, U_2, \ldots, U_{M/B}$. Step 1 can be done in one pass. Each U_i gets partitioned into M/B parts each of length B. Thus there are M/B merging problems, where each problem has to merge M/B sequences each of length B. Since the total number of records in any problem is M, these merging problems can be solved in one pass. Finally, Steps 3 and 4 can also be done in one pass since the length of the dirty sequence is $\leq M^2/B^2 < M$. As a result we have

Lemma 3.2 *$T\left(\frac{M}{B}, M\right) = 3$, if $\frac{M}{B} < \sqrt{M}$.*

3.2 The General Algorithm

Now we are ready to present the general version of the parallel disk sorting algorithm. Here also we will present the algorithm in two cases, one for $\frac{M}{B} \geq \sqrt{M}$ and the other for $\frac{M}{B} < \sqrt{M}$. In either case, initial runs are formed in one pass at the end of which N/M sorted sequences of length M each remain to be merged.

If $\frac{M}{B} \geq \sqrt{M}$, we employ **Algorithm** (l, m)-merge with $l = m = \sqrt{M}$ and $R = \sqrt{M}$. Let K denote $\sqrt{M}$ and let $\frac{N}{M} = K^{2c}$. In other words, $c = \frac{\log(N/M)}{\log M}$. It is easy to see that

$$T(K^{2c}, M) = T(K, M) + T(K, KM) + \cdots + T(K, K^{2c-1}M) \tag{2}$$

The above relation basically means that we start with K^{2c} sequences of length M each; we merge K at a time to end up with K^{2c-1} sequences of length KM each; again merge K at a time to end up with K^{2c-2} sequences of length K^2M each; and so on. Finally we'll have K sequences of length $K^{2c-1}M$ each which are merged. Each of these mergings are done using the **Algorithm** (l, m)-merge with $l = m = \sqrt{M}$.

Let us compute $T(K, K^iM)$ for any i. We have K sequences of length K^iM each. Let these sequences be $U_1, U_2, \ldots, U_K$. In Step 1, each U_j is unshuffled into K parts each of size $K^{i-1}M$. This takes one pass. Now there are K merging problems, where each merging problem involves K sequences of length $K^{i-1}M$ each. The number of passes needed is $T(K, K^{i-1}M)$. In Steps 3 and 4, the length of the dirty sequence is $\leq K^2 = M$. Clearly, this takes only one pass. Therefore,

$$T(K, K^iM) = T(K, K^{i-1}M) + 2.$$

Expanding this out we see,

$$T(K, K^iM) = 2i + T(K, M) = 2i + 3.$$

We have made use of the fact that $T(K, M) = 3$ (c.f. Lemma 3.1).

Note that there are K^{2c-i-1} subproblems of the above kind (i.e., merging K sequences of length K^iM each). Each such subproblem can be solved using $2i + 3$ passes over the data in the subproblem (i.e., $K^{i+1}M$ records). Thus all of these K^{2c-i-1} subproblems can be solved with $2i + 3$ passes over the entire input (of N records).

Substituting this into Equation 2, we get

$$T(K^{2c}, M) = \sum_{i=0}^{2c-1} (2i + 3) = 4c^2 + 4c$$

where $c = \frac{\log(N/M)}{\log M}$. If $N \leq M^3$, the above merging cost is ≤ 24 passes.

We have the following

Theorem 3.1 *The number of read passes needed to sort N records is $1+4\left(\frac{\log(N/M)}{\log M}\right)^2 + 4\frac{\log(N/M)}{\log M}$, if $\frac{M}{B} \geq \sqrt{M}$. This number of passes is no more than $\left[\frac{\log(N/M)}{\log\left(\min\{\sqrt{M}, M/B\}\right)} + 1\right]^2$.*

Now consider the case $\frac{M}{B} < \sqrt{M}$. **Algorithm** (l, m)-merge will be used with $l = m = \frac{M}{B}$ and $R = \frac{M}{B}$. Let Q denote $\frac{M}{B}$ and let $\frac{N}{M} = Q^d$. That is, $d = \frac{\log(N/M)}{\log(M/B)}$. As before we have

$$T(Q^d, M) = T(Q, M) + T(Q, QM) + \cdots + T(Q, Q^{d-1}M) \tag{3}$$

In order to compute $T(Q, Q^iM)$ for any i, note that we have Q sequences of length Q^iM each. Let $U_1, U_2, \ldots, U_Q$ be these sequences. In Step 1, each U_j is unshuffled into Q parts each of size $Q^{i-1}M$. This takes one pass. Now there are Q merging problems. Each merging problem has Q sequences of length $Q^{i-1}M$ each. The number of passes needed to perform all these mergings is $T(Q, Q^{i-1}M)$. Steps 3 and 4 can be performed in one pass since the length of the dirty sequence is $\leq Q^2 < M$. Therefore,

$$T(Q, Q^iM) = T(Q, Q^{i-1}M) + 2.$$

Expansion of this gives

$$T(Q, Q^iM) = 2i + T(Q, M) = 2i + 3.$$

We have made the substitution $T(Q, M) = 3$ (c.f. Lemma 3.2).

Equation 3 now becomes

$$T(Q^d, M) = \sum_{i=0}^{d-1}(2i+3) = d^2 + 2d$$

where $d = \frac{\log(N/M)}{\log(M/B)}$.

Theorem 3.2 *The number of read passes needed to sort N records is upper bounded by $\left[\frac{\log(N/M)}{\log\left(\min\{\sqrt{M}, M/B\}\right)} + 1\right]^2$, if $\frac{M}{B} < \sqrt{M}$.*

Theorems 3.1 and 3.2 readily yield

Theorem 3.3 *We can sort N records in* $\leq \left[\frac{\log(N/M)}{\log\left(\min\{\sqrt{M}, M/B\}\right)} + 1\right]^2$ *read passes over the data, maintaining perfect disk parallelism. In other words, the total number of I/O read operations needed is* $\leq \frac{N}{DB}\left[\frac{\log(N/M)}{\log\left(\min\{\sqrt{M}, M/B\}\right)} + 1\right]^2$.

Observation. In **Algorithm** (l, m)-merge, both l and m have to be $\leq \frac{M}{B}$ in order to achieve perfect disk parallelism.

4 A Comparison of DSM, SRM, and LMM

DSM is not asymptotically optimal. For example, if $M = 2DB$, DSM makes $1 + \log(N/M) = \Omega(\log N)$ passes over the data. If $R = kD$ for some constant k, then SRM is also not asymptotically optimal. Also, under the assumption $M = O(DB)$ (which is the case for LMM), SRM is not asymptotically optimal. However, if $R = kD \log D$, then the expected performance of SRM is optimal. This will mean that the size of M has to be $\Omega(BD \log D)$. On the other hand, LMM is asymptotically optimal assuming that N is a polynomial in M and B is not very large.

Both DSM and LMM are deterministic and need only a reasonable amount of internal memory. Both are very simple and easy to implement. No additional data structures or prefetching techniques are used. Performance analyses are quite simple. On the other hand, SRM is randomized and the performance analysis is more difficult. Its internal memory requirement is $2RB + 4DB + RD$. SRM stores additional (though small) information in each block and maintains a *forecasting data structure*. The analysis of SRM involves the solution of certain occupancy problems.

Now we show that LMM can indeed perform better than DSM when D is large with two examples. A fair comparison of LMM with SRM will require simulations. Since for $R = D$, the number of passes made by SRM is not optimal, we speculate that LMM might compare favorably to SRM when D is large.

In the following examples we won't invoke Theorem 3.3 but rather specialize LMM to get the best possible performance. These examples illustrate that we can get better results than promised by Theorem 3.3. In this context LMM should be thought of as a framework for designing parallel disk sorting algorithms.

Example 4.1 The first example considered is one with $N = 2^{39}; M = 2^{26}; B = 2^{10}$. Here $M/B = 2^{16}$. We need to merge 2^{13} sequences of length 2^{26} each. We apply

a $(2^{13}, 2^{13})$-merge on the sequences. There will be 2^{13} merging problems where each problem involves 2^{13} sequences each of length 2^{13}. Thus the mergings in Step 2 need only one pass. Step 1 takes one pass. Steps 3 and 4 together need one pass. So LMM takes a total of three passes. The number of read passes made by the algorithms for various disk sizes are shown in Table 1. □

N	D	DSM	LMM
2^{39}	2^{15}	13	3
2^{39}	2^{12}	4	3
2^{39}	2^{9}	2	3

Table 1. An Example

Example 4.2 The next example we consider has $N = 2^{42}; M = 2^{26}; B = 2^{10}$. We have to merge 2^{16} sequences each of length 2^{26}. $T(2^{16}, 2^{26}) = T(2^{13}, 2^{26}) + T(2^{3}, 2^{39})$. Both $T(2^{13}, 2^{26})$ and $T(2^{3}, 2^{39})$ can easily be seen to be 3 each using $(2^{13}, 2^{13})$ and $(2^{3}, 2^{16})$-merges, respectively.

Table 2 displays the comparison of DSM and LMM for this example. □

N	D	DSM	LMM
2^{42}	2^{15}	16	6
2^{42}	2^{12}	4	6
2^{42}	2^{9}	3	6

Table 2. Another Example

5 Sparse Enumeration Sort

In this section we present a somewhat simpler algorithm than that of the classical algorithm of Nassimi and Sahni [16] for sparse enumeration sort. The problem is to sort n elements on an $n^{1+1/k}$-processor hypercubic network, for any $k > 0$. The algorithm of [16] has a time bound of $O(k \log n)$. Our algorithm also has the same asymptotic performance.

5.1 Some Definitions

A hypercube of dimension d, denoted $\mathcal{H}_d$ has 2^d processors. Each processor can be labeled with a d-bit binary number. A processor labeled u is connected to those processors with label v such that u and v differ in exactly one bit.

If we fix some i bits and vary the remaining bits of a d-bit binary number, the corresponding processors form a subcube $\mathcal{H}_{d-i}$ in $\mathcal{H}_d$.

5.2 The Algorithm

The main result of this section needs the following

Theorem 5.1 *2^m keys can be sorted on a hypercube of size $n = 2^{2m}$ in $O(\log n)$ time.*

Proof. $\mathcal{H}_{2m}$ can be thought of as consisting of 2^m copies of $\mathcal{H}_m$. Input is given in one of these $\mathcal{H}_m$'s. Broadcast the input so that each $\mathcal{H}_m$ has a copy of the input. This takes $O(\log n)$ time. Each $\mathcal{H}_m$ computes the rank of one input key using the prefix sums algorithm. Prefix computation takes $O(\log n)$ time. Finally route the keys to their sorted positions. This can be done in $O(\log n)$ time as well. □

Theorem 5.2 *We can sort n keys in $O(k \log n)$ time on a hypercube of size $n^{1+1/k}$.*

Proof. Let $S(n)$ be the time needed to sort n keys. Let $T(u, v)$ be the time needed to merge u lists of length v each. Also let $\epsilon = \frac{1}{2k}$.

To begin with group the elements with $n^{1-\epsilon}$ elements in each group. Sort each group recursively in time $S(n^{1-\epsilon})$. Followed by this sort, merge the resultant n^ϵ sorted lists, each of length $n^{1-\epsilon}$, employing **Algorithm** (l, m)-merge with $l = m = n^\epsilon$. Let $Q = n^\epsilon$. Thus it follows that

$$S(n) = S\left(\frac{n}{Q}\right) + T\left(Q, \frac{n}{Q}\right) = S(Q^{2k-1}) + T(Q, Q^{2k-1}) \tag{4}$$

In order to compute $T(Q, Q^j)$, for any j, consider Step 1 of **Algorithm** (l, m)-merge. After unshuffling, each U_i will be partitioned into Q parts with Q^{j-1} elements each. Step 2 will thus take time $T(Q, Q^{j-1})$. Unshuffling in Step 1 and shuffling in Step 3 take time $O(\log Q)$. In Step 4 the length of the dirty sequence is $\leq Q^2$ and hence Step 4 can be completed in $O(\log Q)$ time as well (employing Theorem 5.1).

As a result, we have $T(Q, Q^j) = T(Q, Q^{j-1}) + O(\log Q)$. Expanding this out, we get $T(Q, Q^j) = O(j \log Q)$. Substituting this back in Equation 4, we see that

$$S(n) = S(Q) + \sum_{j=1}^{2k-1} O(j \log Q)$$

$$= O(\log Q) + O(k^2 \log Q) = O(k \log n) \quad \square$$

6 Columnsort As A Special Case

Columnsort has been employed on various parallel models of computing [14, 15]. This algorithm can be described as follows. Let $k_1, k_2, \ldots, k_n$ be the n given numbers. These numbers are thought of as forming a (tall and thin) matrix M with r rows and s columns.(with $r \geq s^2$). There are 7 steps in the algorithm: 1) Sort the columns in increasing order; 2) Transpose the matrix preserving the dimension as $r \times s$. I.e., pick the elements in column major order and fill the rows in row major order; 3) Sort the columns in increasing order; 4) Rearrange the numbers applying the reverse of the permutation employed in step 2; 5) Sort the columns in a way that adjacent columns are sorted in reverse order; 6) Apply two steps of odd-even transposition sort to the rows; and 7) Sort each column in increasing order. At the end of step 7, it can be shown that, the numbers will be sorted in column major order.

Columnsort is thus easily seen to be a special case of LMM where $l = m = s$ (with $n \geq s^3$). In Step 1 of columnsort, the input is grouped into s parts and each part is sorted (c.f. Step 1 of **Algorithm** LMM). In Step 2, the sorted subsequences are s-way unshuffled (c.f. Step 1 of **Algorithm** (l, m)-merge). In Step 3, the unshuffled subsequences are sorted instead of being recursively merged (c.f. Step 2 of **Algorithm** (l, m)-merge). In Step 4 shuffling is performed (c.f. Step 3 of **Algorithm** (l, m)-merge). Steps 5, 6, and 7 of columnsort perform Step 4 of **Algorithm** (l, m)-merge.

Note that the crucial differences between LMM and the columnsort are: 1) LMM is recursive and the columnsort is not; and 2) The columnsort has $l = m$ but LMM is general. In fact in Example 4.2, we have $l \neq m$.

It is also noteworthy that Kunde's algorithm for the mesh [13] somewhat resembles the columnsort algorithm.

7 Conclusions

We have introduced a new sorting algorithm called the (l, m)-mergesort. An application of this algorithm to the parallel disk sorting problem yields an asymptotically optimal algorithm that performs better than DSM and possibly SRM algorithms when the number of disks is large. This algorithm is as simple as DSM requiring no nontrivial data structures or prefetching techniques. We strongly believe that our algorithm will perform well in practice as evidenced in the recent work of Pearson [20]. We also believe that applications of our sorting scheme to other models will result in similar performances. We have shown that the columnsort of Leighton is a special case of (l, m)-mergesort and applied LMM to the hypercube.

Acknowledgements

This research was supported in part by an NSF Award CCR-95-03-007 and an EPA Grant R-825-293-01-0. The author is thankful to Tom Cormen and Matt Pearson for many stimulating discussions. The author also thanks the reviewers for their thoughtful comments.

References

[1] A. Aggarwal and C. G. Plaxton, Optimal Parallel Sorting in Multi-Level Storage, *Proc. Fifth Annual ACM Symposium on Discrete Algorithms*, 1994, pp. 659-668.

[2] A. Aggarwal and J. S. Vitter, The Input/Output Complexity of Sorting and Related Problems, *Communications of the ACM* 31(9), 1988, pp. 1116-1127.

[3] M. Ajtai, J. Komlós and E. Szemerédi, An $O(n \log n)$ Sorting Network, *Proc. 15th Annual ACM Symposium on Theory of Computing*, 1983, pp. 1-9.

[4] L. Arge, The Buffer Tree: A New Technique for Optimal I/O-Algorithms, *Proc. 4th International Workshop on Algorithms and Data Structures (WADS)*, 1995, pp. 334-345.

[5] K. Batcher, Sorting Networks and their Applications, *Proc. AFIPS Spring Joint Computing Conference* 32, 1968, pp. 307-314.

[6] R. Barve, E. F. Grove, and J. S. Vitter, Simple Randomized Mergesort on Parallel Disks, *Parallel Computing* 23(4-5), 1997, pp. 601-631.

[7] R. Cole, Parallel Merge Sort, *SIAM Journal on Computing* 17(4), 1988, pp. 770-785.

[8] T.E. Cormen and M. Hirschl, Early Experiences in Evaluating the Parallel Disk Model with the ViC* Implementation, *Parallel Computing* 23, 1997, pp. 571-600.

[9] R.E. Cypher and C.G. Plaxton, Deterministic Sorting in Nearly Logarithmic Time on the Hypercube and Related Computers, *Proc. ACM Symposium on Theory of Computing*, 1990, pp. 193-203.

[10] M. Dowd, Y. Perl, L. Rudolph, and M. Saks, The Periodic Balanced Sorting Network, *Journal of the ACM* 36, 1989, pp. 738-757.

[11] E. Horowitz, S. Sahni, and S. Rajasekaran, *Computer Algorithms*, W. H. Freeman Press, 1998.

[12] D. Knuth, *Searching and Sorting*, *The Art of Computer Programming*, Volume 3, Addison-Wesley, 1973.

[13] M. Kunde, Block Gossiping on Grids and Tori: Deterministic Sorting and Routing Match the Bisection Bound, *Proc. First European Symposium on Algorithms*, 1993, pp. 272-283.

[14] T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays–Trees–Hypercube*, Morgan-Kaufmann Publishers, 1992.

[15] T. Leighton, Tight Bounds on the Complexity of Parallel Sorting, *IEEE Transactions on Computers* C34(4), 1985, pp. 344-354.

[16] D. Nassimi and S. Sahni, Parallel Permutation and Sorting Algorithms and a New Generalized Connection Network, *Journal of the ACM* 29(3), 1982, pp. 642-667.

[17] M. H. Nodine and J. S. Vitter, Greed Sort: Optimal Deterministic Sorting on Parallel Disks, *Journal of the ACM* 42(4), 1995, pp. 919-933.

[18] M. H. Nodine and J. S. Vitter, Large Scale Sorting in Parallel Memories, *Proc. Third Annual ACM Symposium on Parallel Algorithms and Architectures*, 1990, pp. 29-39.

[19] V. S. Pai, A. A. Schaffer, and P. J. Varman, Markov Analysis of Multiple-Disk Prefetching Strategies for External Merging, *Theoretical Computer Science* 128(2), 1994, pp. 211-239.

[20] M.D. Pearson, Fast Out-of-Core Sorting on Parallel Disk Systems, Technical Report PCS-TR99-351, Computer Science Department, Dartmouth College, 1999, `ftp://ftp.cs.dartmouth.edu/TR/TR99-351.ps.Z`.

[21] S. Rajasekaran, Sorting and Selection on Interconnection Networks, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 21, 1995, pp. 275-296.

[22] J.H. Reif and L.G. Valiant, A Logarithmic Time Sort for Linear Size Networks, *Journal of the ACM* 34(1), 1987, pp. 60-76.

[23] R. Reischuk, Probabilistic Parallel Algorithms for Sorting and Selection, *SIAM Journal of Computing* 14(2), 1985, pp. 396-409.

[24] C.D. Thompson and H.T. Kung, Sorting on a Mesh Connected Parallel Computer, *Communications of the ACM* 20(4), 1977, pp. 263-271.

[25] J. S. Vitter and E. A. M. Shriver, Algorithms for Parallel Memory I: Two-Level Memories, *Algorithmica* 12(2-3), 1994, pp. 110-147.